Design patterns

General patterns for common problems in object–oriented programming

Author:
Martin Pitt
email: martin@piware.de
Home: http://www.piware.de

August 21, 2000

This document was created after the book “Design Patterns”, by Grady Booch et al.

Contents

Chapter 1

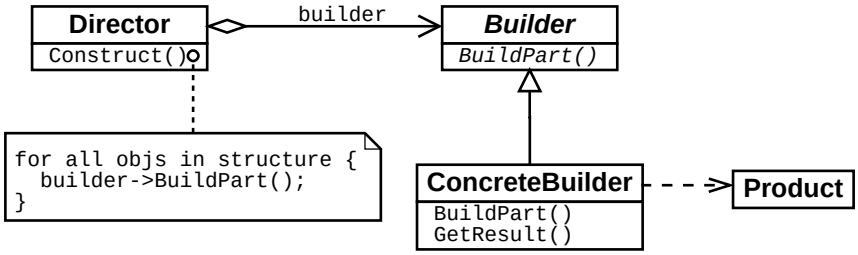
Creational patterns

1.1 Builder

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structure



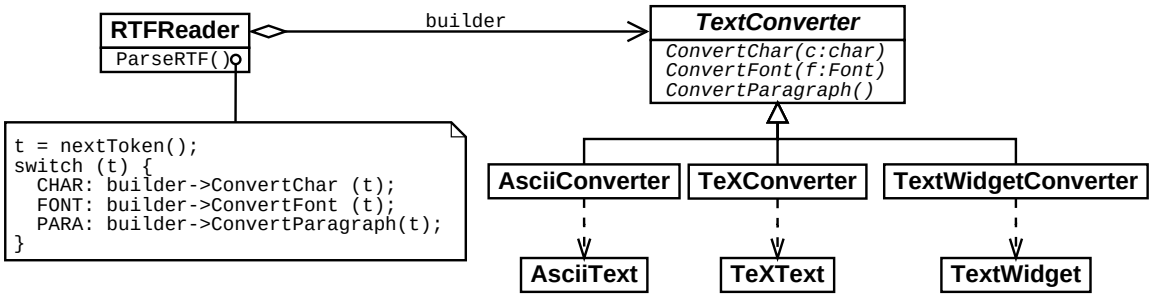
Applications

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
- the construction process must allow different representations for the object that is constructed

Consequences

- lets you vary a product's internal representation
- isolates code for construction and representation
- finer control over construction process $\rightarrow$ step by step

Example

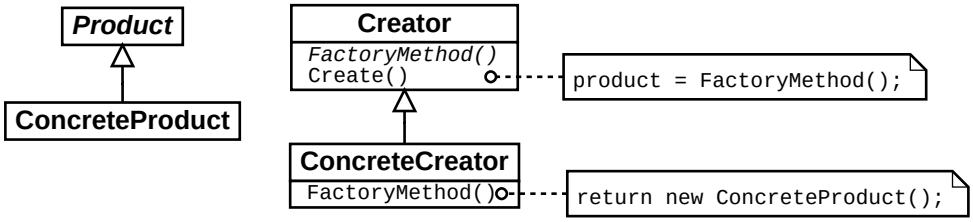


1.2 Factory Method / Virtual constructor

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Structure



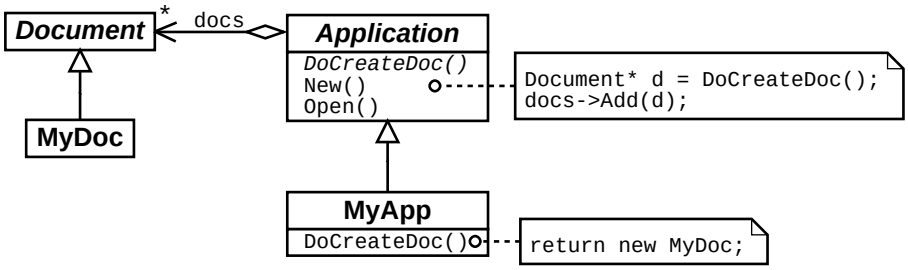
Applications

- a class can not anticipate the class of objects it must create
- a class wants its subclasses to specify toe object it creates
- classes delegate responsibility to one of several helper subclasses and you want to localize the knowledge of which helper subclass is the delegate

Consequences

- provides hooks for subclasses
- connects parallel class hierarchies (called by client)

Example

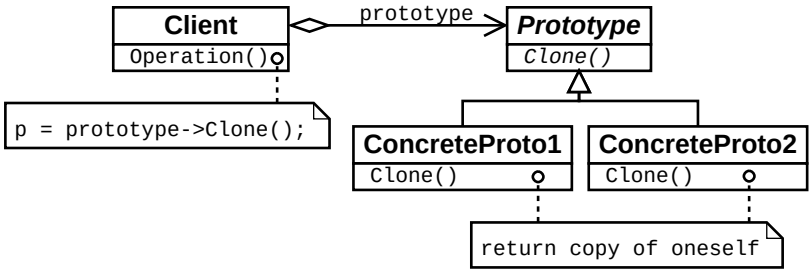


1.3 Prototype

Intent

Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

Structure



Applications

- System should be independent of how its prototypes are created, composed, or represented
- when the classes to instantiate are specified at runtime, e. g. by dynamic loading
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products
- when instances of a class can have only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually each time with the appropriate state.

Consequences

- adding and removing products at run–time
- specifying new objects by changing values $\rightarrow$ new “classes” without programming
- reduced subclassing $\rightarrow$ do not need no creator at all (Factory Method $\rightarrow$ parallel creator hierarchy)
- configure an application with classes dynamically

Example

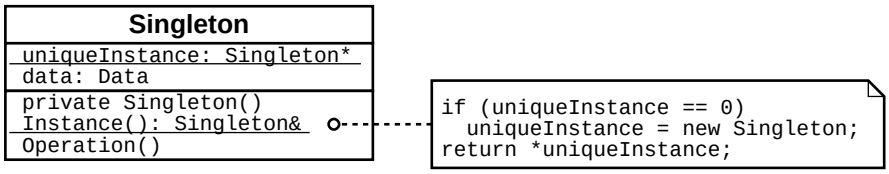
object I/O over binary channels

1.4 Singleton

Intent

Ensure that a class has only one instance, and provide a global point of access to it.

Structure



Consequences

- controlled access to sole instance
- reduced name space (no global variables)
- permits refinement of operations and representation (subclassable)
- permits variable number of instances (with slight changes)
- more flexible than class operations (static members can not be virtual, and can not have multiple instances)

Chapter 2

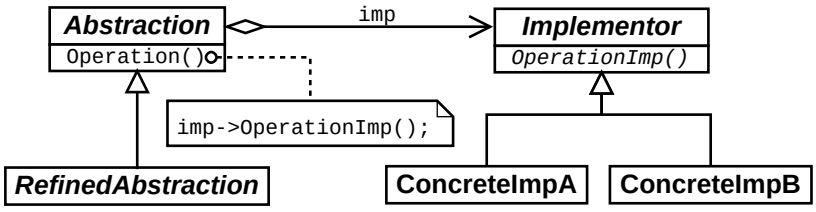
Structural patterns

2.1 Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Structure



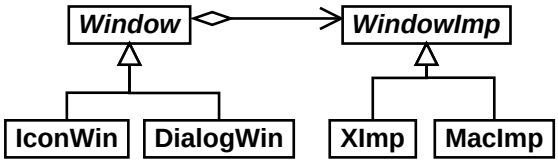
Applications

- You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, e. g. when the implementation must be selected or switched at run–time.
- Both the abstraction and the implementation should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- You want to hide implementation of an abstraction completely from clients.
- You have a proliferation of classes (e. g. `Window` $\rightarrow$ `MotifIconWindow`, `XDialogWindow`, etc.). Such a class hierarchy indicates the need for splitting an object in two parts.
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from clients.

Consequences

- Decoupling interface and implementation
- Improved extensibility (abstraction and implementation hierarchies are extendable independently)
- Hiding implementation details from clients

Example

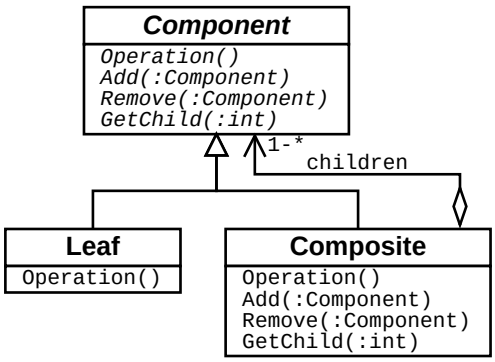


2.2 Composite

Intent

Compose objects into tree structures to represent part–whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.

Structure



Consequences

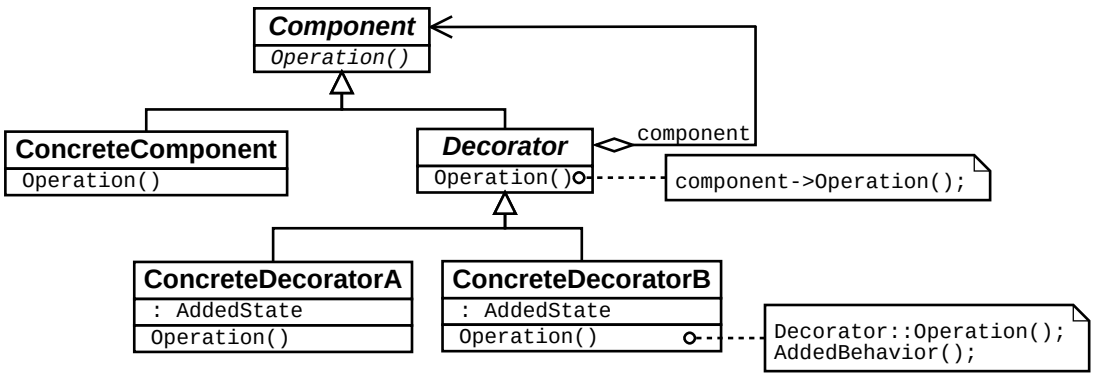
- Defines dynamic hierarchy consisting of primitive and composite objects. Primitive objects can be composed into more complex ones, which in turn can be composed, and so on. Wherever a client expects a primitive, it can also take a composite object.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally do not know (and should not care) whether they are dealing with a leaf or a composite component.
- Makes it easier to add new kinds of components. Clients work automatically with new subclasses.
- ! Can make your design overly general, because it is hard to restrict components of a composition. → You have to use run–time checks.

2.3 Decorator / Wrapper

Intent

Attach additional responsibilities to an object dynamically. Decorator provides a flexible alternative to subclassing for extending functionality.

Structure



Applications

- to add responsibility to individual objects dynamically and transparently, that is, without affecting other objects
- for responsibilities that can be withdrawn
- when extension by subclassing is impractical (support every possible combination)

Consequences

- more flexibility than static inheritance
- avoids feature–laden classes high up in the hierarchy → "pay as you go"
- ! decorator and its component are not identical → you should not rely on object identity
- ! lots of little objects → differ only in the way they are interconnected

Example

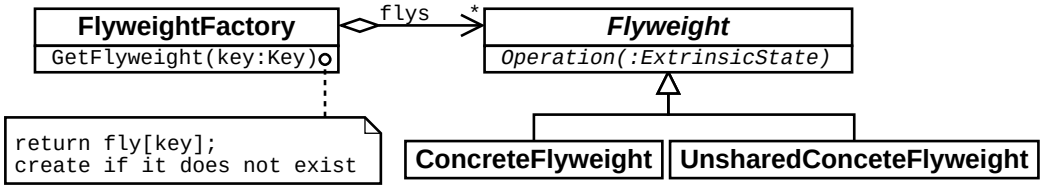
`TextView` → `ScrollDecorator`, `BorderDecorator`

2.4 Flyweight

Intent

Use sharing to support large numbers of fine–grained objects effectively.

Structure



Applications

- An application uses a large number of objects.
- Storage costs are high, because of the sheer quantity of objects.
- Most object state can be made extrinsic (state that depends on context and can be computed $\leftrightarrow$ intrinsic state).
- The application does not depend on object identity.

Consequences

- storage savings
- ! higher runtime costs (computing and transfer of extrinsic state)

Example

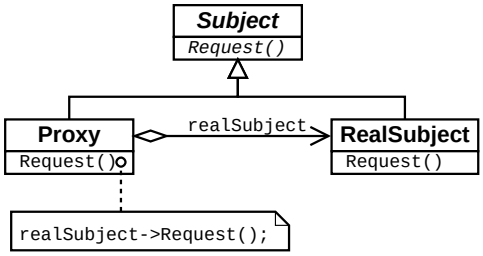
Editor $\rightarrow$ each character is represented by a separate object (intrinsic state: char code, extrinsic state: position)

2.5 Proxy / Surrogate

Intent

Provide a surrogate or placeholder for another object to control access to it.

Structure



Applications

- Remote proxy: provides a local representative for an object in a different address space
- Virtual proxy: creates expensive objects on demand
- Protection proxy: controls access to the original object
- Smart reference: additional actions if object is accessed (instead of the bare pointer)

Consequences

- Remote proxy: can hide the fact that an object resides in a different address space
- Virtual proxy: perform optimizations like create–on–demand
- Protection proxy and Smart Reference: additional housekeeping
- Copy–on–write with shared objects (needs reference counting)

Example

graphic image in a document editor has a “visible” flag; shall only be shown if visible, but should be treated uniformly

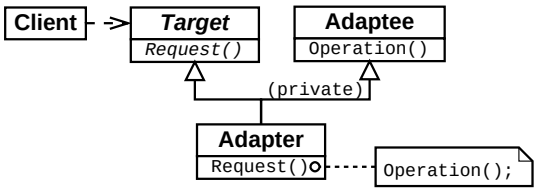
2.6 Adapter / Wrapper

Intent

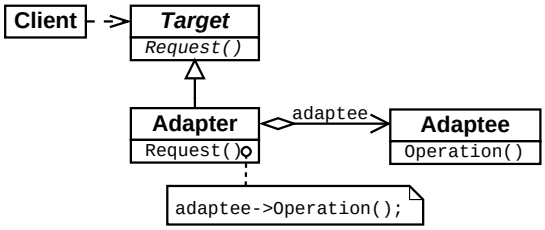
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Structure

Class adapter:



Object adapter:



Consequences

Class adapter: easy to change behavior (by overriding), but does not work with subclasses
Object adapter: can use several existing subclasses, but it is harder to override behavior

Chapter 3

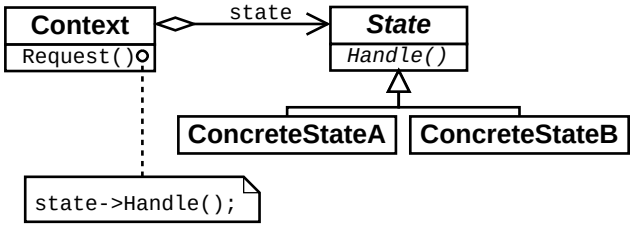
Behavioral patterns

3.1 State

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Structure



Applications

- An object's behavior depends on its state, and it must change its behavior at run–time depending on that state
- Operations have large, multi-part conditional statements that depend on the object's state. The State pattern puts each branch in a separate class

Consequences

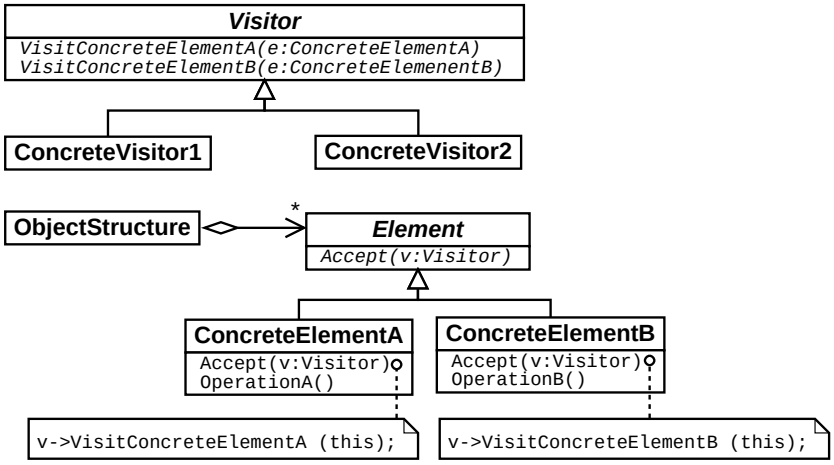
- It localizes style–specific behavior and partitions behavior for different states.
- Makes state transitions explicit.
- State objects can be shared (Flyweight).

3.2 Visitor

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure



Applications

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class.
- When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface of all visitors which is potentially costly. If the classes change often, then it is probably better to define operations in those classes.

Consequences

- Visitor makes adding new operations easy
- a Visitor gathers related operations and separates unrelated ones
- visiting across class hierarchies; objects do not even have a common parent class
- accumulating state; no global variables or extra state parameters
- ! adding new concrete element classes is hard

Example

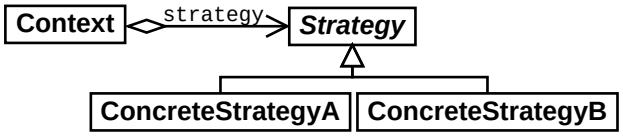
`Glyph` → `Bitmap`, `Character`, `Composite`: spell checking only with characters, without dynamic casting

3.3 Strategy / Policy

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure



Applications

- Many related classes differ only in their behavior. Strategy provides a way to configure a class with one of many behaviors.
- You need different variants of an algorithm, e. g. for different space/time trade–offs.
- An algorithm uses data clients should not know about.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own strategy class.

Consequences

- families of related algorithms
- alternative to subclassing (vary algorithms independently)
- Strategy eliminates conditional statements
- choice of implementations
- ! client must be aware of different strategies (or default strategy)
- ! communication overhead between Strategy and Context
- ! increased number of objects

Example

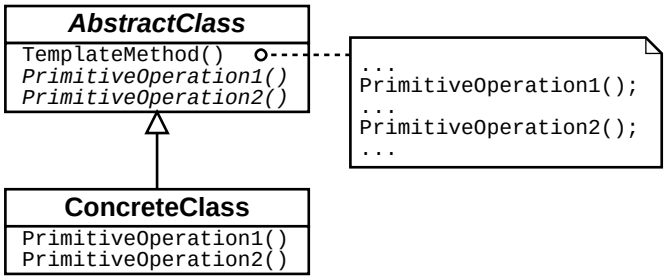
Line breaking algorithm: fast (per–line), TeX (per paragraph), array, ...

3.4 Template Method

Intent

Define the skeleton of an algorithm in an operation, defering some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure (provide hook operations at certain points for subclasses).

Structure



Applications

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- Common behavior among subclasses should be factored out and localized in a common class to avoid code duplication
- to control subclass extensions: you define a Template Method that calls hook operations at specific points, thereby permitting extensions only at those points (Template Method is non–virtual!)

Consequences

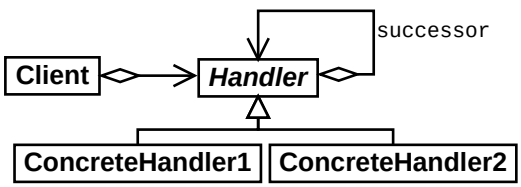
- fundamental technique for code reuse
- mean for factoring out common behavior
- inverted control structure (Hollywood principle: "don't call us, we call you")

3.5 Chain of responsibility

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Structure



Applications

- More than one object many handle the request, and the handler is not known in advance. The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Consequences

- reduced coupling
- added flexibility in assigning responsibilities to objects
- ! receipt is not guaranteed; request can go unhandled

Example

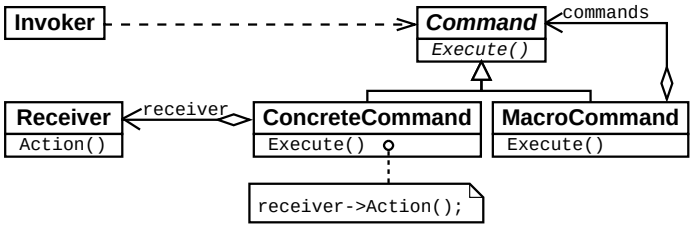
Help system: Button → Dialog → Application; if no specific help is available, use parent (more general) one

3.6 Command / Action

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo/redo.

Structure



Applications

- parameterize objects by an action to perform (e. g. menu item) → object–oriented replacement for callbacks
- specify, queue, and execute requests at different times; a command object can have a lifetime independent of the original request
- support undo by introducing an `Unexecute` method and saving previous state if necessary; commands are stored in a history list that can be traversed back (undo) and forward (redo)
- support logging changes (by augmenting interface with load/store)
- structure a system around high–level operations built on primitive operations (encapsulation of a set of data to change)

Consequences

- decouples invoker from object knowing how to perform it
- you can assemble commands into a composite command for e. g. macros
- it is easy to add new commands, because you do not have to change existing classes

Example

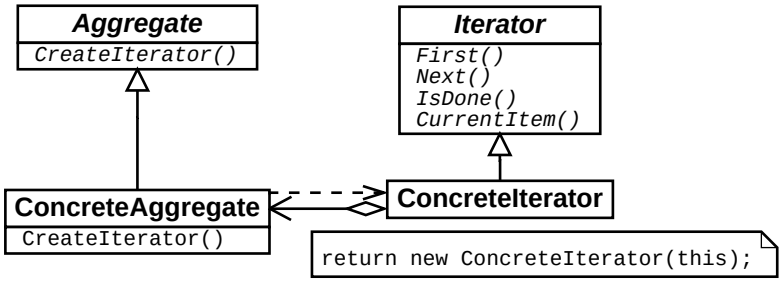
commands in a menu, also yieldable by a button, etc.

3.7 Iterator

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Structure



Applications

- support multiple traversals of aggregate objects, more than one at a time
- provide a uniform interface for traversing different aggregate structures

Consequences

- supports variations in the traversal of an aggregate
- Iterator simplifies the Aggregate interface

Types

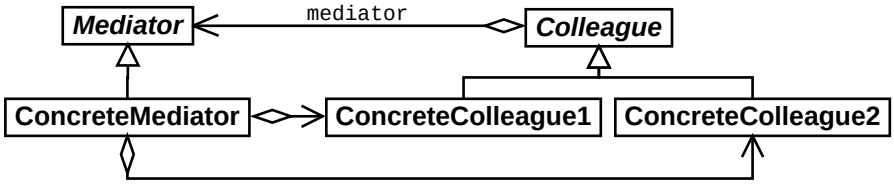
- External iterator: client controls iteration (`for` loop)
- Internal iterator: Iterator traverses in one shot (easier, but unflexible)
- Cursor: Iterator only points to current element, Aggregate defines traversal algorithm → `Aggregate::Next (Cursor)`
- Robust iterator: ensures that insertions and removals will not interfere traversals without copying the aggregate

3.8 Mediator

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Structure



Applications

- A set of objects communicates in well-defined, but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many others.
- Behavior that is distributed between several classes should be customizable without a lot of subclassing.

Consequences

- limits subclassing
- decouples colleagues
- simplifies object protocols
- abstracts how objects cooperate
- ! centralizes control $\rightarrow$ monolithic, hard to maintain

Example

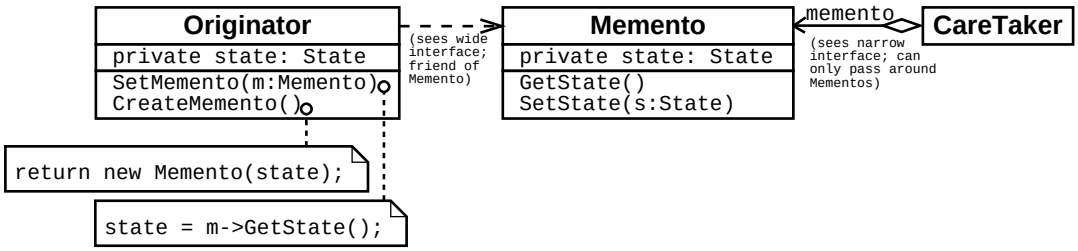
director in file select box where a list box causes an input line to change or a button enable/disable $\rightarrow$ use Mediator and standard stock classes

3.9 Memento / Token

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Structure



Applications

- a snapshot of (some portion of) an object's state must be saved so that it can be restored, AND
- a direct interface to obtain the state would expose implementation details and break encapsulation

Consequences

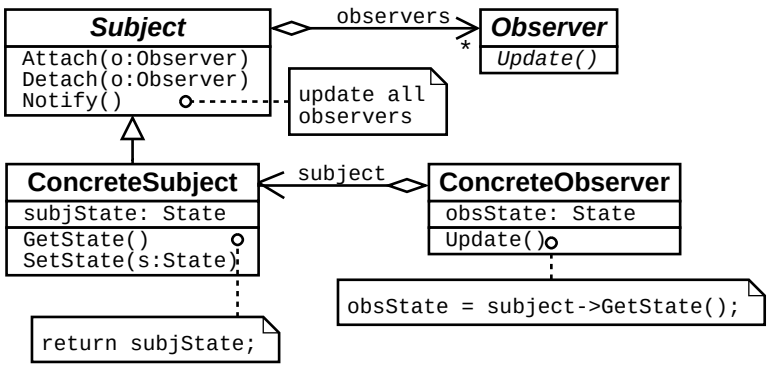
- preserves encapsulation boundaries
- simplifies Originator
- ! using Memento may be expensive
- ! may be difficult in some languages to define narrow / wide interface
- ! hidden costs in caring for Mementos

3.10 Observer / Publish–Subscribe

Intent

Define a one–to–many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

Structure



Applications

- When an abstraction has two aspects, one depending on the other, encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object should be able to notify others, and you do not know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who those objects are. In other words, you do not want these objects to be tightly coupled.

Consequences

- abstract coupling between Subject and Observers
- support for broadcast communication
- ! unexpected updates (simple protocol does not determine *what* changed → superfluous updates?)

Example

- numerical data → bar and pie-slice spreadsheet diagram at the same time
- clock timer → `DigitalClock`, `AnalogClock`